behaviac 使用手册

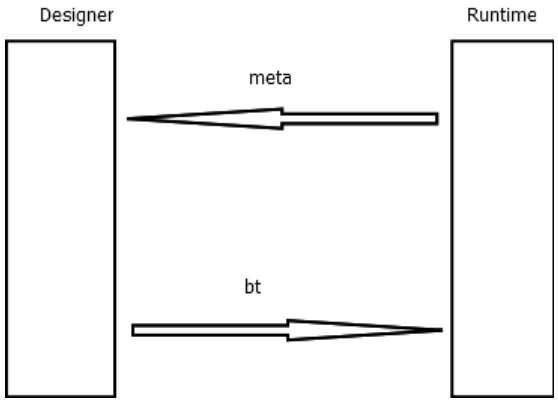
腾讯游戏
Tencent Games

https://github.com/TencentOpen/behaviac

目录

1 概述

behaviac 中间件是我们对行为树的一种实现方案，包括一个编辑器（Designer）和一个运行时库（Runtime）。其中，编辑器用来编辑和调试行为树，运行时库用来导出元信息给编辑器并解释和执行编辑器生成的行为树，如下图所示。



编辑器只能运行在 Windows 平台上，运行时库支持 C++和 C#语言两个版本，对 Unity 引擎原生支持 C#，运行时库目前支持全平台，包括 Windows/Linux/Android/iOS 等。

该组件的使用场景，支持但不限于游戏中的逻辑、角色的人工智能、动画的控制等方面。

2 安装

安装 behaviac 套件的系统配置需求如下所示：

最低配置：

硬件环境：

CPU：Intel core2 Duo 2.0G 或同等配置；内存：2GB 以上；显卡：nvidia 6600GT 或同等配置。

软件环境：

操作系统：Windows XP/Vista/7/8；屏幕分辨率：1024*768

推荐配置：

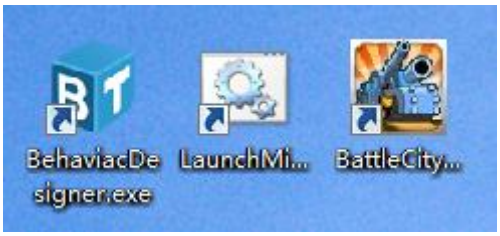
硬件环境：

CPU：Intel I5 2.93G 或更高配置；内存：4GB 以上；显卡：nvidia GTS450 或更高配置。

软件环境：

操作系统：Windows 7/8；屏幕分辨率：1680*1050

运行 BehaviacSetup_***.exe 安装包文件，选择安装路径（推荐直接使用默认路径），安装 behaviac 相关套件。安装完成后，会在桌面生成 3 个快捷方式，如下图所示：



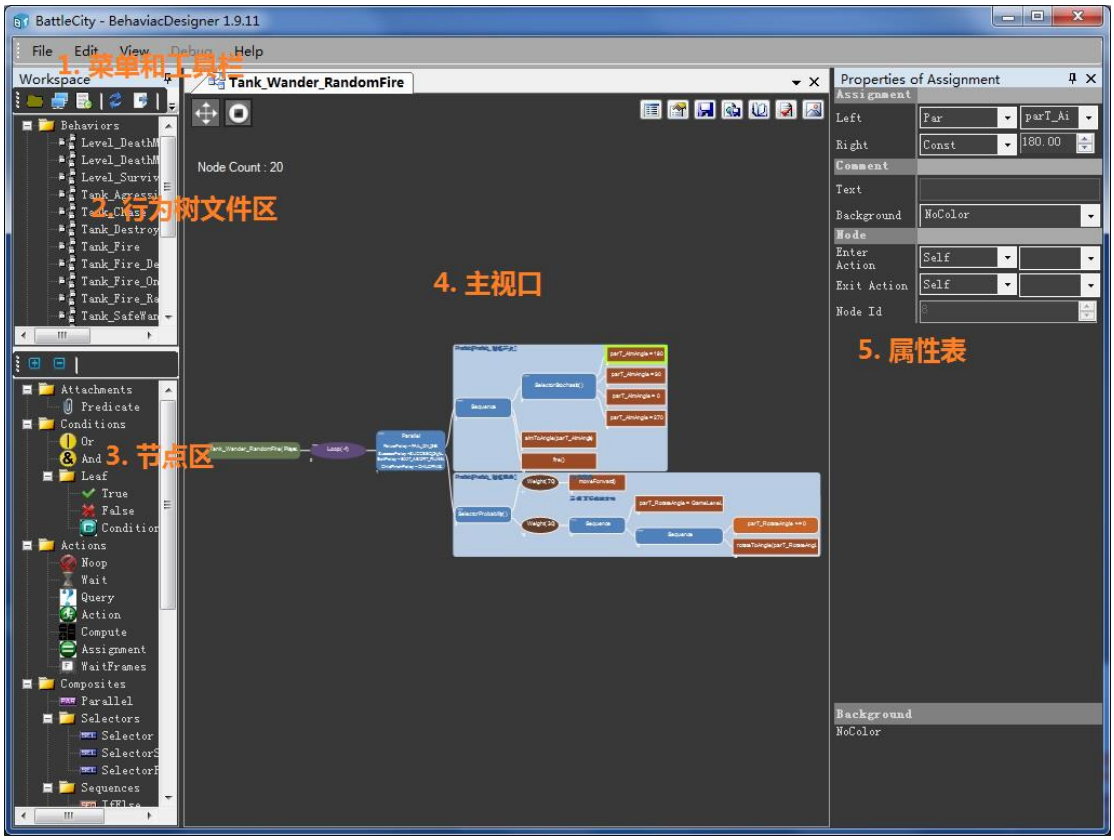
其中，最左边的图标为 behaviac 编辑器，中间的图标为采用 C++实现的小游戏 Ship（用到了 behaviac 的 C++运行时库），右边的图标为采用 Unity 引擎实现的小游戏 Tank（用到了 behaviac 的 C#运行时库）。

3 使用

3.1 基本介绍

编辑器首次打开后为英文版的界面，若需更改为中文版，则可以依次选择菜单项“File”->“Settings”，在弹出的 Settings 对话框中，设置“Language”为“Chinese”后，点击“OK”，退出并重新打开编辑器，则转为中文版界面。

编辑器主要分为几个部分：菜单和工具栏、行为树文件区、节点区、主视口和属性表等，如下图所示：



1. **菜单和工具栏**：主要有新建/打开工作区、新建/保存/导出/关闭行为树文件、设置、

退出、帮助等按键。

2. **行为树文件区**：列出了当前打开的工作区里面的所有行为树文件，双击某个行为树文件节点可以打开该文件。
3. **节点区**：给出了 behaviac 支持的所有行为树节点类型，在编辑行为树的时候，可以通过鼠标拖拽添加到主视口的行为树上去。
4. **主视口**：显示了当前打开的行为树图形，可以打开多个行为树，分页显示。
5. **属性表**：在主视口中点击选择某个节点后，属性表会更新显示当前选中节点的所有属性，可以编辑各个属性。

注意：所有窗口的停靠位置均可根据个人喜好，通过鼠标在窗口边框的拖拽重新布局。**编辑器中鼠标、键盘的按键介绍，请查看菜单中的“帮助”->“控制说明”。**

此外，在主视口的左上角，有个 2 个快捷按钮，如下图所示：

左边的按钮表示 Normal 和 Pan 模式的切换，编辑器启动后默认为 Normal 模式。点击选择激活为 Pan 模式后（或者按住 Alt 键），鼠标左键拖拽整个行为树，右键缩放整个行为树。

右边的按钮用于居中显示整个行为树。

编辑器中其他的按钮均可将鼠标停在上面查看其 tips，便可知其功能。

编辑器分为三种操作模式：

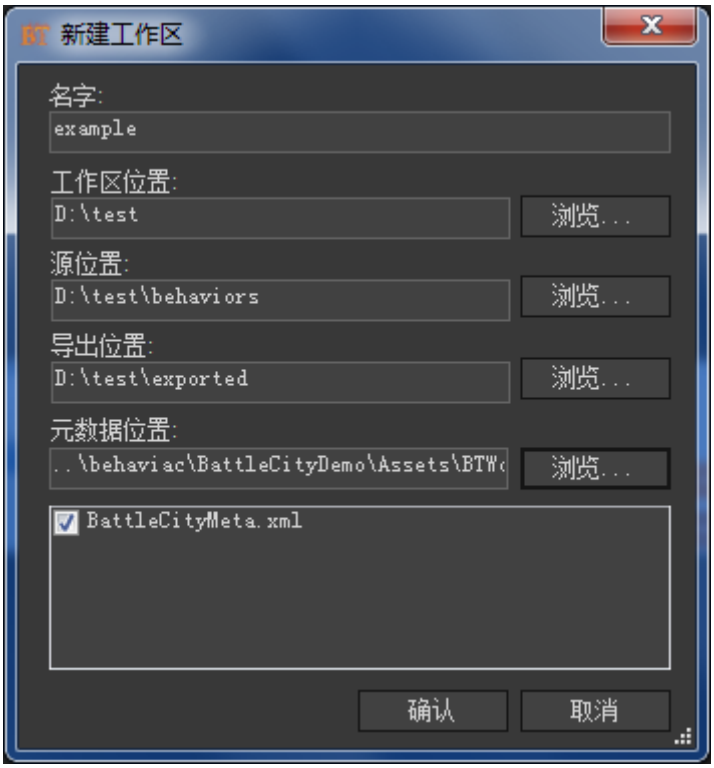
- **编辑模式**：这是常规模式，编辑器启动后默认进入该模式，主要用于为游戏项目编辑所需的行为树。
- **连接模式**：这是编辑器连接上游戏之后进入的模式，此时不能编辑行为树本身，只能用于观察行为树的高亮执行路径、设置断点等调试相关的操作。
- **分析模式**：这是离线模式，在断开游戏时保存了连接过程中的所有消息，在该模式可以模拟连接模式游戏端发送过来的消息，进行离线分析行为树的执行情况。

3.2 操作说明

3.2.1 新建工作区

首次使用编辑器，需要为自己的项目创建一个工作区：

1. 点击菜单“文件”->“新建工作区”，弹出新建工作区对话框，如下图所示。
2. 设置工作区的名字、位置（即工作区的保存路径）、源位置（即原始行为树的保存路径）、导出位置（即运行时库所需行为树的导出路径）和元数据位置（即运行时库导出的元信息文件路径）。
3. 源位置是存放编辑器创建的行为树的位置。导出位置是存放导出的行为树的位置。源位置和导出位置缺省情况下位于工作区所在的目录内：behaviors 和 exported。
4. 元数据位置、源位置和导出位置需要和工作区文件必须位于同一个盘符下（如果位于不同的盘符下，调试的时候会有问题）。缺省情况下，它们位于同一个顶级目录下。



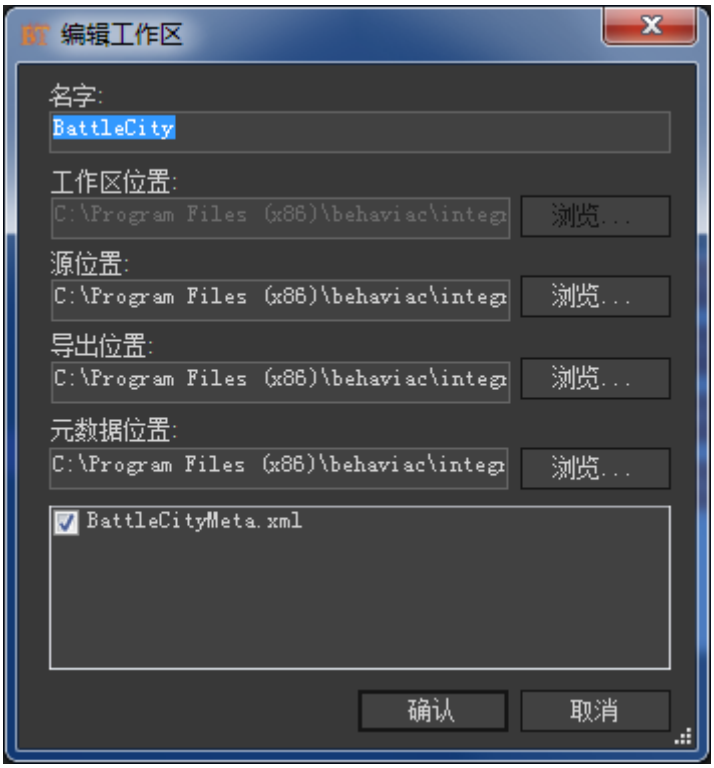
3.2.2 打开工作区

点击菜单“文件”->“打开工作区”，选择保存在磁盘上的*.workspace.xml 文件。

在安装路径下有提供的例子供参考，例如默认安装路径下的“C:\Program Files (x86)\behaviac\integration\BattleCityDemo\Assets\BTWorkspace\BattleCity.workspace.xml”，或者从菜单中的“文件”->“最近打开的工作区”列表中选择最近打开过的工作区。

3.2.3 编辑工作区

点击菜单“文件”->“编辑工作区”，可以修改该工作区的设置（名字、元数据位置、源位置、导出位置等），如下图所示：



3.2.4 新建并编辑行为树文件

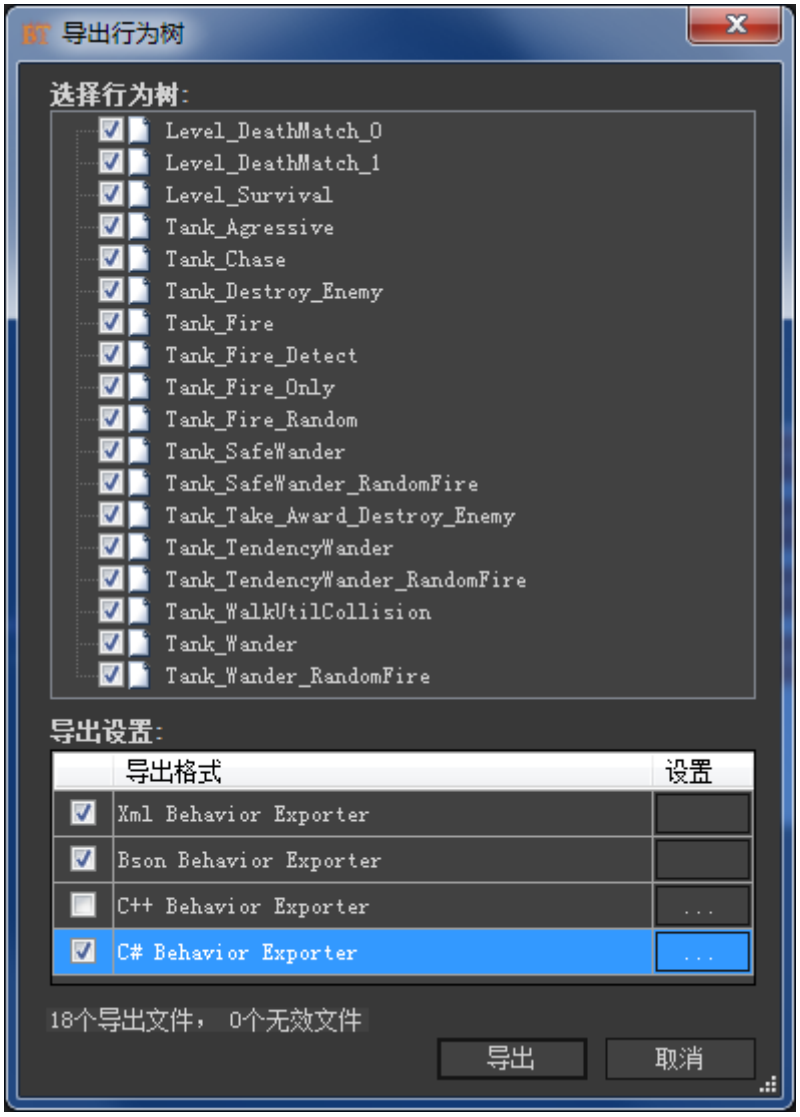
1. 在打开的工作区中，从工具栏里面点击“新建行为树”按钮或者通过菜单中“文件”->“新建行为树”项，创建一个行为树文件。
2. 在主视口中，为新建的行为树根节点选择设置“Agent 类型”。
3. 在“节点区”中用鼠标左键选择并拖拽一些节点构建需要的行为树，并设置每个节点的属性。每种节点的具体用法请参考帮助文档，菜单项“帮助”->“概述”。
4. 编辑过程中，支持 Undo/Redo、保存等文件操作。

3.2.5 打开行为树文件

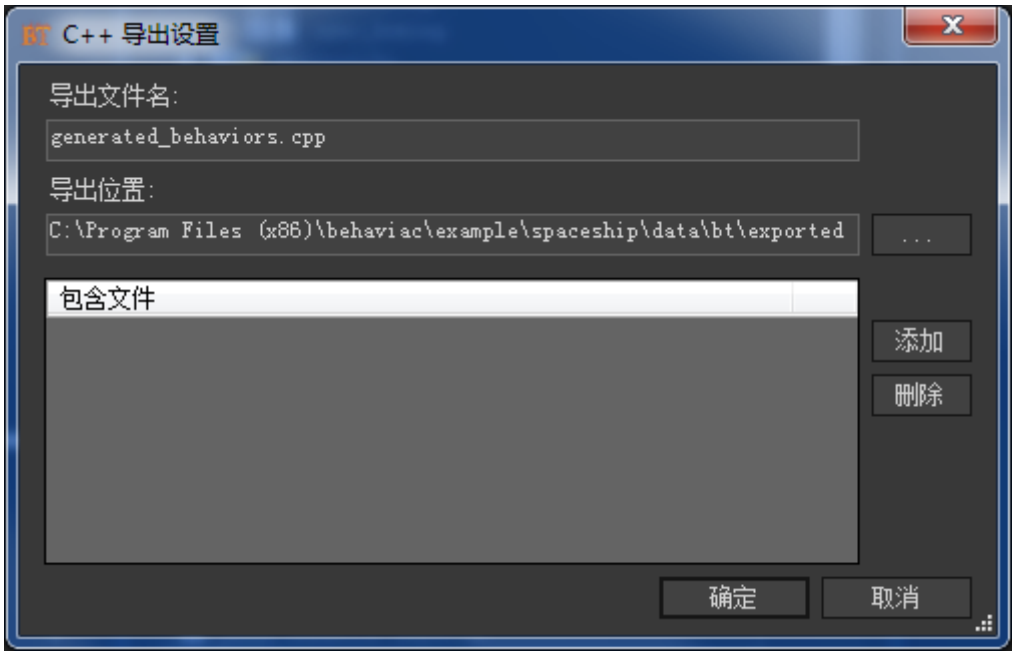
在“工作区”的行为树文件列表中，双击某个行为树文件节点就可以在主视口中打开该文件。

3.2.6 导出行为树文件

1. 编辑完行为树之后，可以通过菜单项“文件”->“导出行为树”（或者快捷键 Ctrl+T）导出当前打开的行为树文件。
2. 通过菜单中“文件”->“导出全部”项（或者快捷键 Ctrl+Shift+T）导出整个工作区里面的所有行为树文件。
3. 目前支持 4 种文件格式的导出，即 XML、BSON、C++和 C#，如下图所示：



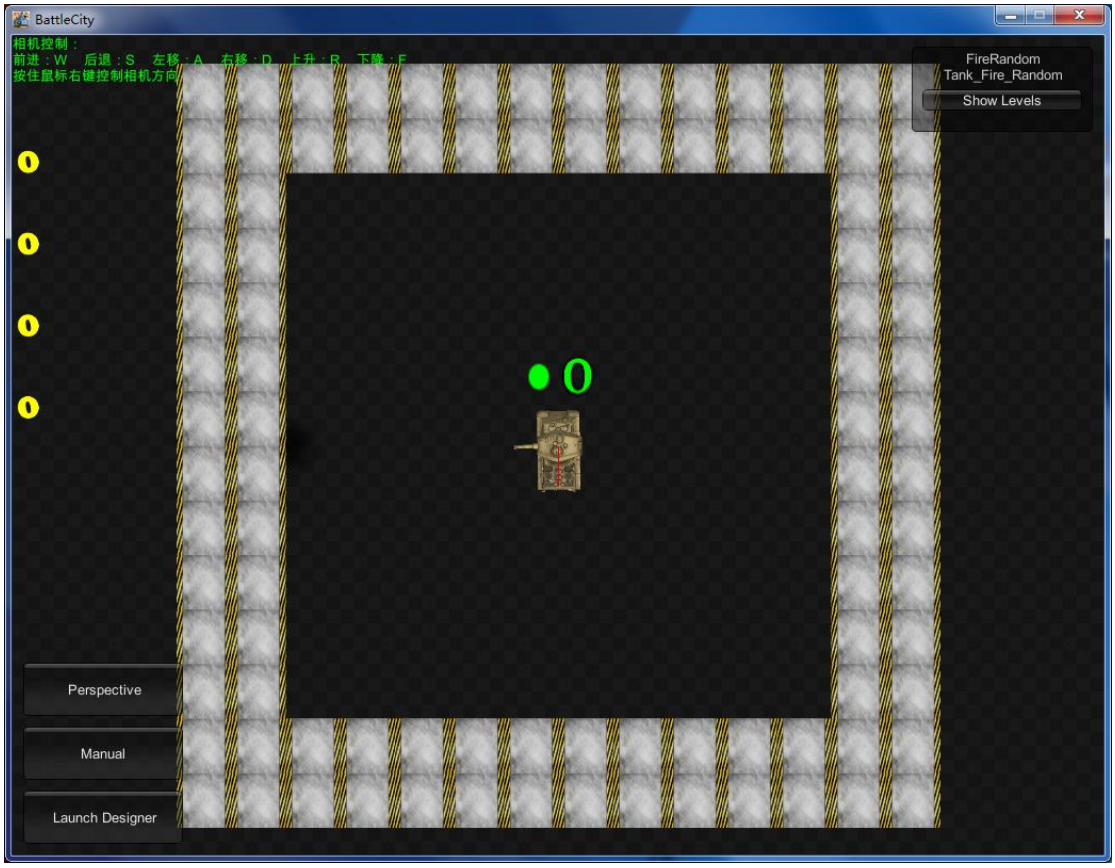
4. 对于 C++和 C#格式的导出，还可以设置导出的文件名和位置。点击上图“设置”列中的“…”按钮，可以弹出“C++/C#导出设置”对话框。对于 C++文件，还需添加游戏项目中游戏类的.h 头文件，如下图所示：



5. 运行时库支持行为树导出文件的热加载，如果在编辑器修改完行为树之后，重新导出，游戏端会自动重新加载所用到的行为树，无需重启游戏。

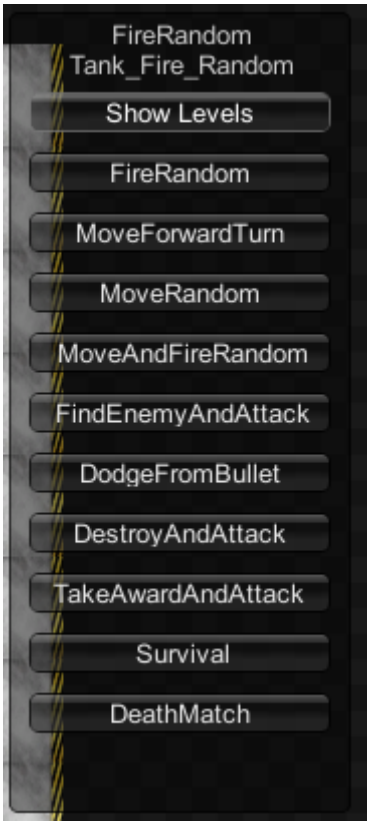
3.2.7 连接游戏

1. 双击桌面快捷方式，启动小游戏 BattleCityDemo，如下图所示：



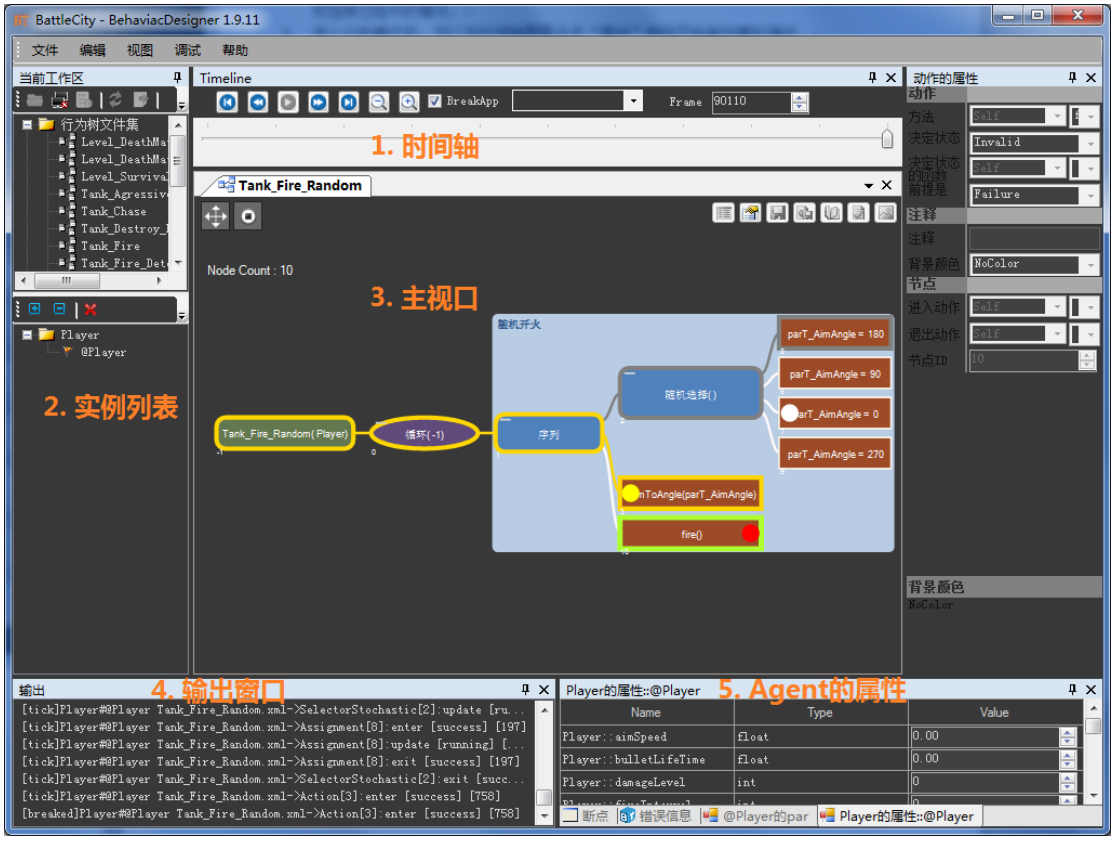
https://github.com/TencentOpen/behaviac

2. 点击游戏主界面右上角的“Show Levels”按钮，可以切换不同的关卡，如下图所示：



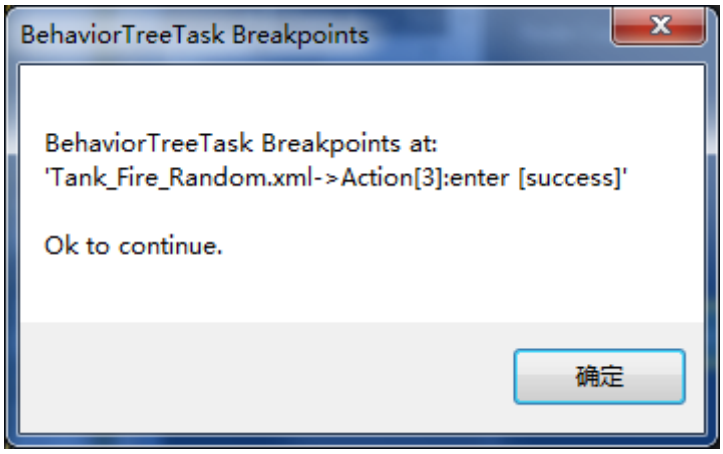
3. 点击游戏主界面左下角的“Launch Designer”按钮，启动编辑器，编辑器会自动打开 BattleCity 的工作区，并打开游戏当前关卡所用到的行为树（如果没有切换其他关卡，默认打开的是 Tank_Fire_Random 行为树文件）。
4. 点击编辑器工具栏中的“连接游戏”按钮或通过菜单项“文件”->“连接游戏”（快捷键 Ctrl+L），开始连接游戏。默认的，一般无需更改服务器 IP 和端口号，如下图所示：

5. 连接成功后，编辑器从编辑模式切换到连接模式，如下图所示：



1. **时间轴（Timeline）**：默认位于编辑器最上方，用于表示当前帧相关的信息。
2. **实例列表**：列出了游戏中所有的对象实例，双击实例节点后可以开始跟踪调试选中的实例，右键单击实例节点弹出如下菜单。其中“调试”跟双击节点的作用一致，开始跟踪选中的实例，主视口会高亮显示行为树的执行路径；“查看 par”用于跟踪当前选中实例的所有 par 值的变化，编辑器会弹出实例的 par 列表；“查看属性”用于跟踪当前选中实例的所有属性值的变化，编辑器会弹出实例的属性列表。

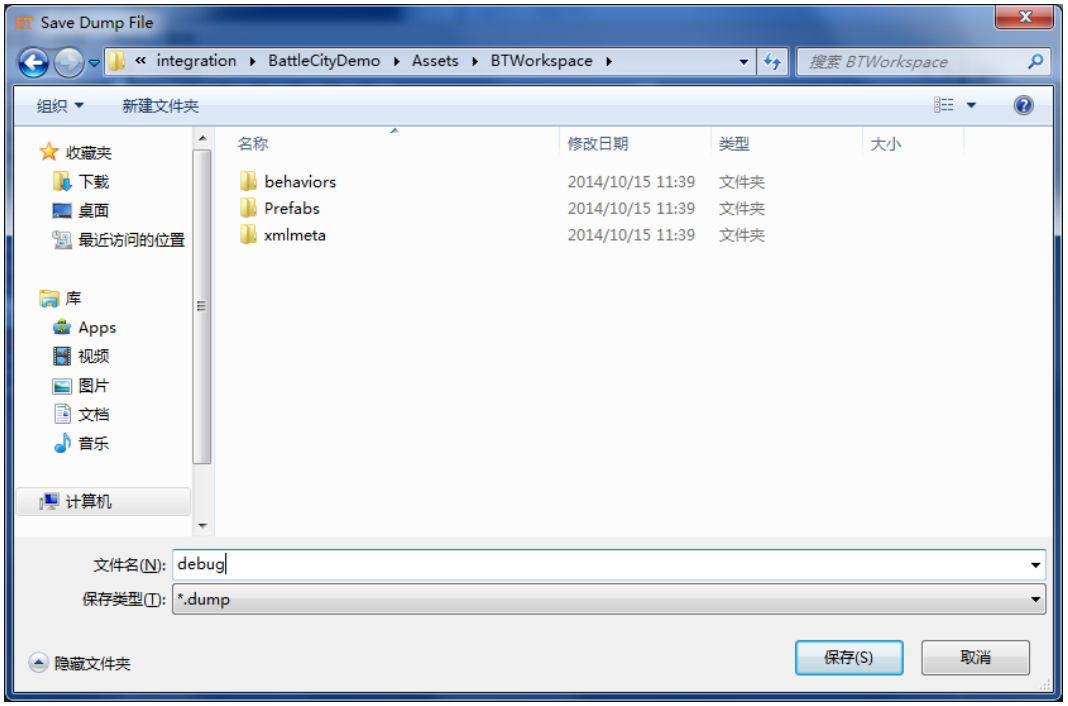
3. **主视口**：高亮的节点边框和连接线段，表示程序当前执行的路径情况。双击树中节点的左右两侧（左侧表示 enter，右侧表示 exit），可以设置节点的断点，用于游戏在断点位置停下来，游戏断下来后会弹出下图所示的对话框：



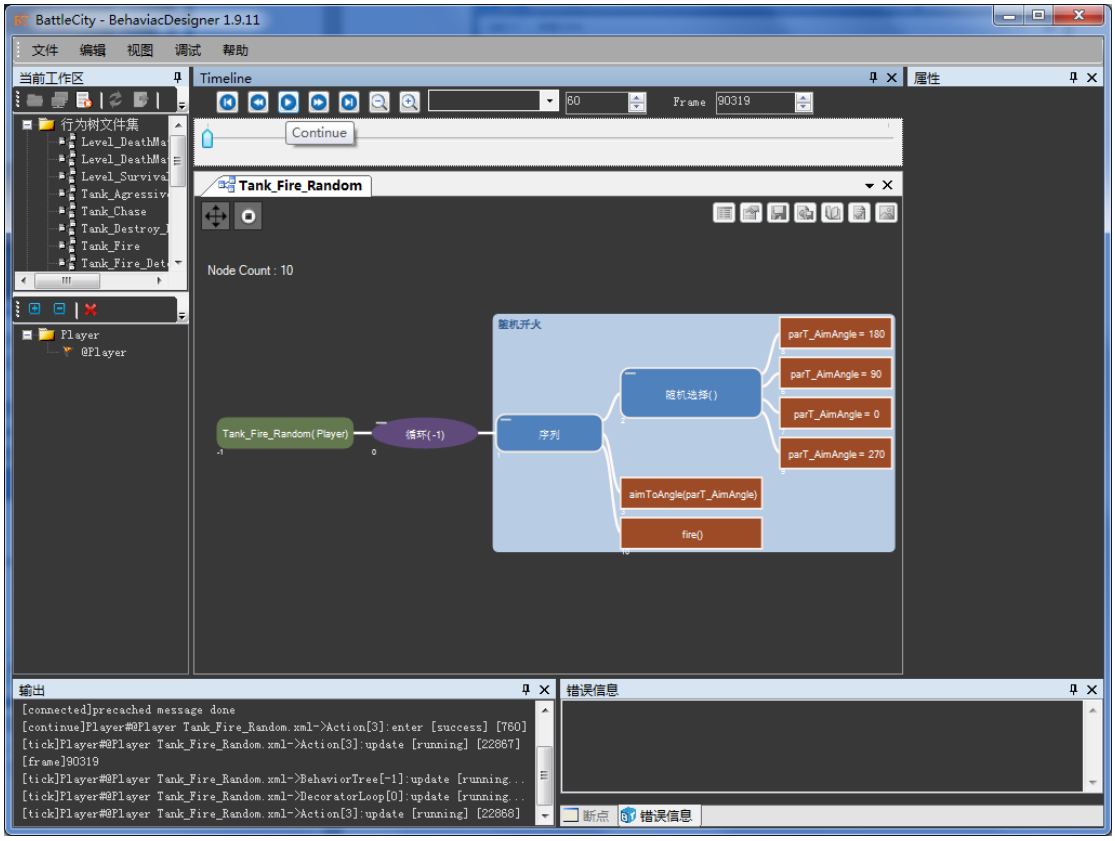
4. **输出窗口**：列出了收到的全部消息，用于调试时查看记录。
5. **Agent 的属性**：可以显示当前 Agent 属性的变化。

3.2.8 分析导出数据

1. 点击菜单或工具栏中的“断开游戏”按键，断开与游戏端的连接，回到编辑模式。
2. 默认弹出如下对话框，保存之前连接模式下调试的相关数据。



3. 点击菜单或工具栏中的“分析导出文件”按键，选择打开上一步导出的数据文件，编辑器进入到分析模式。
4. 分析模式的界面布局跟连接模式类似，但是可以通过时间轴里面的播放按钮开始重现连接过程中的情况，如下图所示：



5. 要退出分析模式，可以点击菜单或工具栏中的“终止分析”按键，回到编辑模式。

3.2.9 其他

除了 BattleCity 小游戏（基于 Unity 引擎实现，运行时库为 C#版）之外，还有一个 Ship 小游戏（C++实现，运行时库为 C++版）。

双击桌面图标启动 Ship 游戏后，跟编辑器的连接时，需要单独启动编辑器，然后再点击编辑器工具栏中的“连接游戏”按钮或通过菜单项“文件”->“连接游戏”（快捷键 Ctrl+L），开始连接游戏。连接后，执行情况跟 BattleCity 类似。

4 注册、导出和使用元信息

行为树中间件的基本运作机制就是在 C++库（Lib）中注册并导出供行为树引擎和编辑器（Designer）使用的 XML 元信息。其中，XML 元信息里面包含了 C++库的反射信息（类自身的描述及其属性与方法），以及各种实例（单件类或者类的多个实例）等。

行为树编辑器根据 C++库导出的元信息，可以编辑所选择的某个 Agent 实例的行为树，在行为树的每个节点上可以根据需要选择或编辑属性、方法及其参数等。

整个工作流程主要分为以下几个步骤：

4.1 注册元信息

1. 在.h 文件中，根据项目需要按以下步骤编写自己的游戏类：
   1. 该类需要从 **behaviac::Agent** 基类继承。
   2. 首先用宏 **DECLARE_BEHAVIAC_OBJECT** 声明该类及其父类，用于行为树引擎内部的反射系统所需的类型信息。
   3. 为该类添加必要的属性和方法等。

   如下代码样例所示：

```
class AgentBase : public behaviac::Agent
{
public:
    DECLARE_BEHAVIAC_OBJECT(AgentBase, behaviac::Agent);

    void        method1()       { }
    float       method2(int n)  { return 0.0f; }
    static int  static_method3()  { return 0; }

private:
    int             property1;
    bool            property2;
    float           property3;
    static Tag::string_t  sProperty4;
};
```

2. 在.cpp 文件中，通过一系列宏注册该类自身的描述及其属性与方法：
   1. 宏 **BEGIN_PROPERTIES_DESCRIPTION** 和 **END_PROPERTIES_DESCRIPTION** 表示类型信息注册的开始和结束。
   2. 宏 **CLASS_DISPLAYNAME** 和 **CLASS_DESC** 用于注册类自身的显示名和描述。
   3. 宏 **REGISTER_PROPERTY** 用于注册类的属性，可以通过**.DISPLAYNAME** 的追加方式为属性添加显示名，通过**.DESC** 的追加方式为属性添加描述。

4. 宏 **REGISTER_METHOD** 接受 1 个参数用于注册类的方法，可以通过**.DISPLAYNAME** 的追加方式为方法添加显示名，通过**.DESC** 的追加方式为方法添加描述，通过**.PARAM_INFO** 的追加方式为参数添加显示名，描述。如果参数类型是数字类型（int，unsigned int，float 等），**.PARAM_INFO** 还可以用来指定参数的有效范围。
5. 宏 **REGISTER_EVENT** 用于注册命名的事件。REGISTER_EVENT 可以接受 1 到 4 个参数。第 1 个参数必须是该事件的名字，一个字符串，之后的参数是数据类型，指定该事件的参数类型。

如下代码样例所示：

```
BEGIN_PROPERTIES_DESCRIPTION(AgentBase)
    CLASS_DISPLAYNAME(L"Agent基类")
    CLASS_DESC(L"Agent基类的说明")

    REGISTER_PROPERTY(property1);
    REGISTER_PROPERTY(property2).DISPLAYNAME(L"属性2");
    REGISTER_PROPERTY(property3).DESC(L"属性说明3");
    REGISTER_PROPERTY(sProperty4).DISPLAYNAME(L"属性4").DESC(L"属性说明4");

    REGISTER_METHOD(method1).DISPLAYNAME(L"方法1").DESC(L"方法说明1");
    REGISTER_METHOD(method2).PARAM_INFO (L"n", L"参数", 1, 10);
    REGISTER_METHOD (static_method3);

    REGISTER_EVENT("EventDead").SETSTATIC();
    REGISTER_EVENT("Exploded", int).PARAM_INFO(L"半径", L"半径参数的说明");
END_PROPERTIES_DESCRIPTION()
```

4.2 导出元信息

注册完类信息之后，按以下步骤导出元数据文件：

1. 在初始化函数里面，添加 **Agent::Register**<AgentBase>()用于注册类信息到引擎库中。
2. 调用 **Agent::ExportMetas**(“agents.xml”)导出为 agents.xml 的元数据文件。
3. 在释放函数里面，添加 **Agent::Unregister**<AgentBase>()用于释放类型的注册信息。

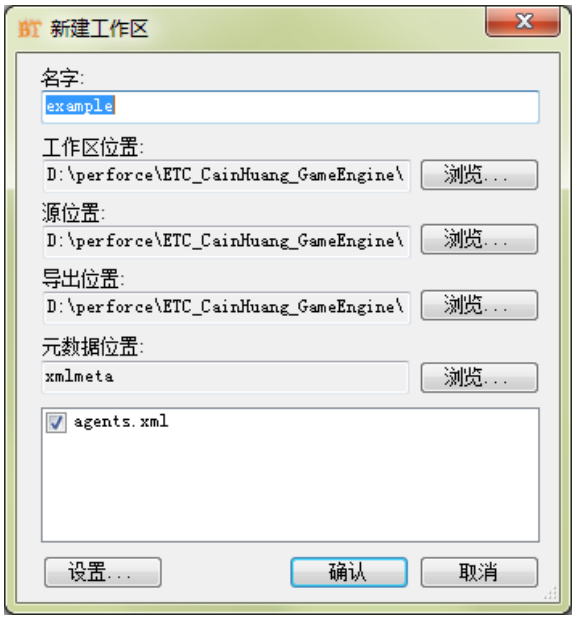
如下代码样例所示：

```
Agent::Register<AgentBase>();

Agent* pA = Agent::Create<AgentBase>();
Agent::ExportMetas("../xmlmeta/agents.xml");
BEHAVIAC_DELETE(pA);

Agent::Unregister<AgentBase>();
```

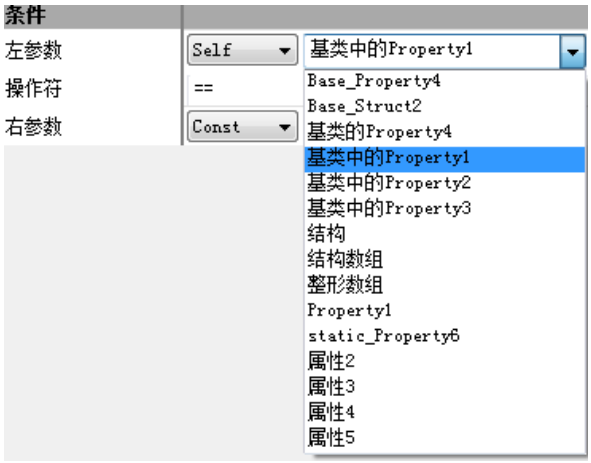
更多细节可以参考行为树中间件源码库中 btunittest 工程的 behaviortest.h 和 behaviorRegister.cpp 文件中 AgentBase 类相关的代码。

4.3 使用元信息

1. 在行为树编辑器中新建一个工作区，并根据上一步骤导出的元数据文件所在的文件夹，设置该工作区的“元数据位置”，可以看到 agents.xml 文件已经可以使用，如下图所示：



2. 新建行为树文件，并为该行为树添加一些节点，选择某个节点后，可以设置该节点的属性（可选属性正是来至于上面导出的元信息），如下图所示：

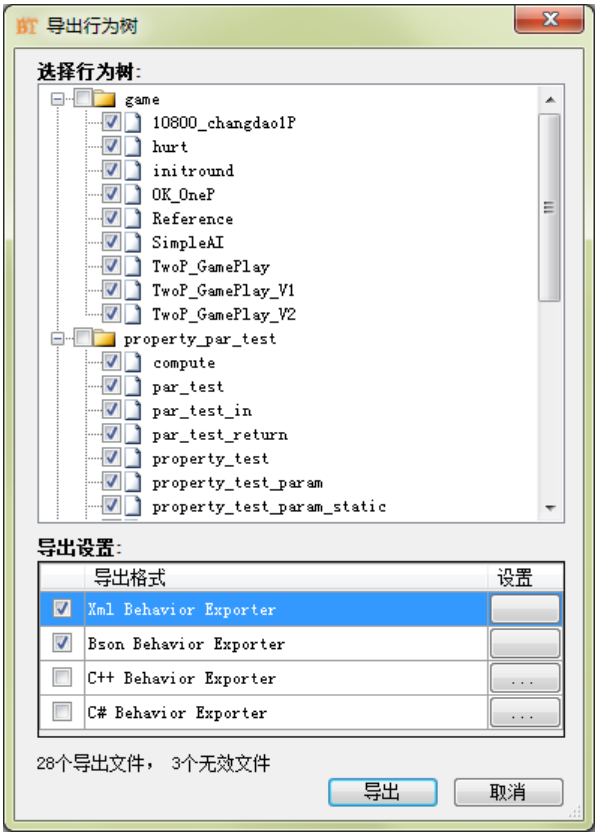


5 导出行为树

目前 behaviac 支持 4 种文件格式（XML、BSON、C++和 C#）行为树的导出。在项目开发过程中，建议使用 XML 格式的导出文件，以便于调试查错等；而在最终的 release 版本中，只需导出 C++或 C#格式的行为树文件，以便提高运行效率。

5.1 导出和使用 XML/BSON 行为树

1. 在编辑器中导出整个工作区文件，在“导出行为树”对话框中，选择“Xml Behavior Exporter”或“Bson Behavior Exporter”，如下图所示：



2. 导出行为树文件结束，回到 C++应用程序代码端（C#类似），按以下步骤开始使用行为树文件：
   2.1 在初始化函数中，添加 **Agent::Register**<AgentBase>()注册类信息到引擎库中。
   2.2 调用 **BehaviorTree::SetWorkspaceSettings**()函数，该函数有两个参数，第一个参数设置行为树文件的加载路径，第二个参数为加载行为树的文件格式。

2.3 通过调用 Agent 的接口 **btload**()加载所需的行为树文件。

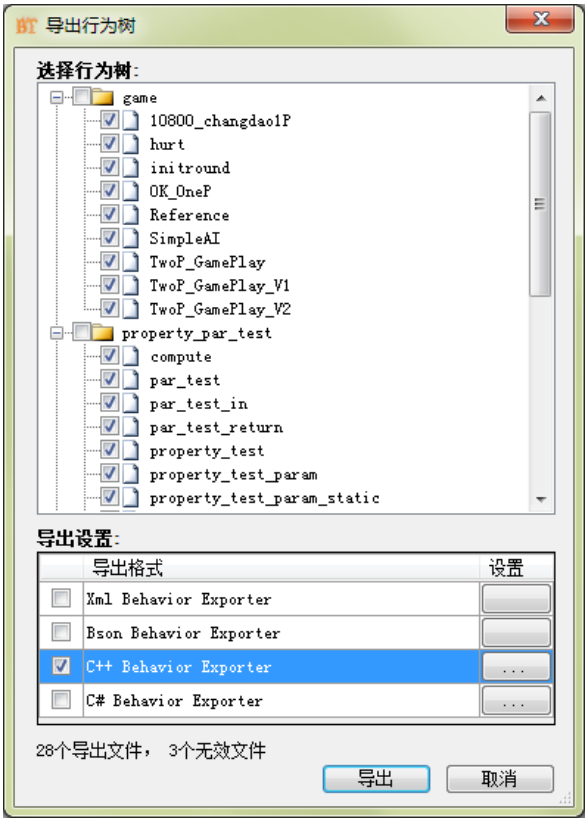
2.4 在释放函数里面，添加 **Agent::Unregister**<AgentBase>()用于释放类型的注册信息。

如下代码样例所示：

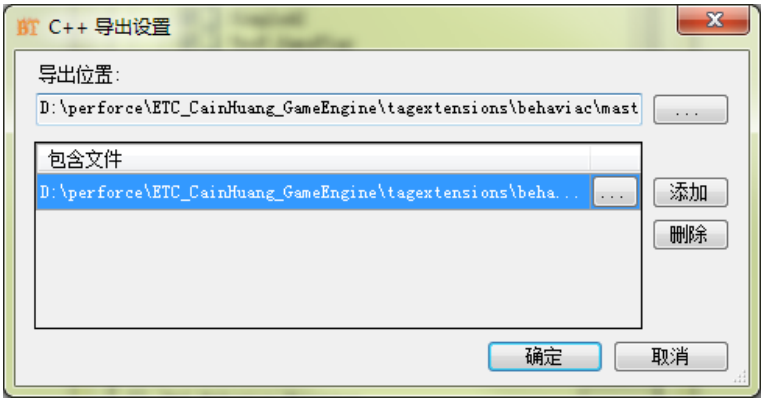
```
Agent::Register<AgentBase>();

Agent* pA = Agent::Create<AgentBase>();
BehaviorTree::SetWorkspaceSettings("../exported/" , BehaviorTree::EFF_xml);
pA->btload("bt.xml");

BEHAVIAC_DELETE(pA);
Agent::Unregister<AgentBase>();
```

5.2 导出和使用 C++行为树

1. 在“导出行为树”对话框中，选择“C++ Behavior Exporter”，如下图所示：



2. 点击上图中右侧的“…”设置按钮，在弹出的“C++导出设置”对话框中设置生成文件所在的位置，并可以添加项目中游戏类（从 Agent 类派生而来）所在的.h 头文件，添加的头文件将会被包含在生成的.cpp 文件中，如下图所示：



2.1 回到“导出行为树”对话框，点击“导出”按钮，开始导出.cpp 文件。在指定的导出位置（默认为当前工作区的 exported 文件夹）会自动生成一个名为 generated_behaviors.cpp 的文件，里面生成了在 editor 中添加的所有行为树类。

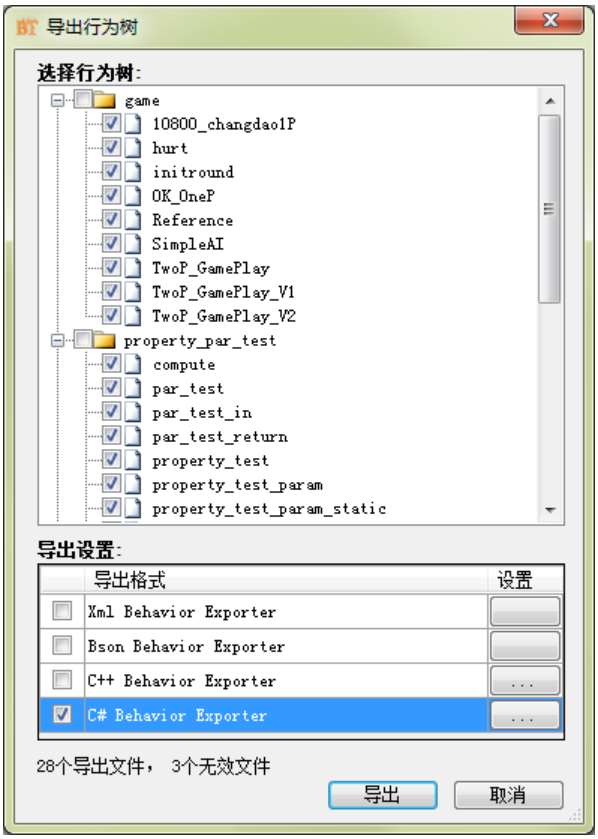
2.2 在项目中 include 该 generated_behaviors.cpp 文件，就可以与前面提及的 XML 行为树文件一样的接口和方式加载使用该.cpp 文件。

例如，在 btunittest 工程的 btunittest.cpp 文件中，有如下代码：

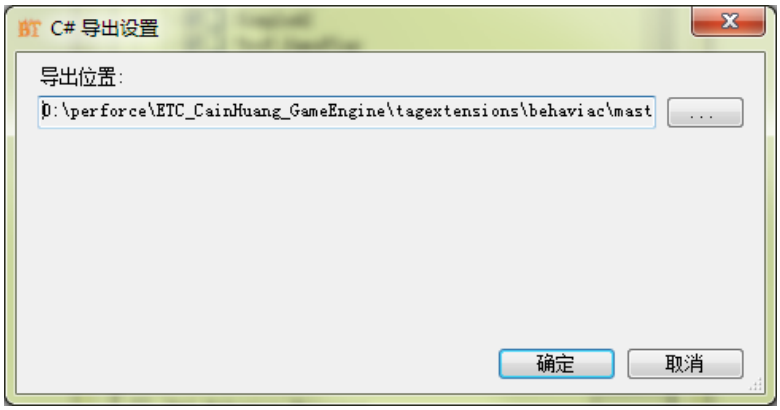
```
#include ".. /example_workspace/exported/generated_behaviors.cpp"

Agent* pA = Agent::Create<AgentTest>();
AgentTest* pAgent = AgentTest::DynamicCast(pA);

BehaviorTree::SetWorkspaceSettings("../example_workspace/exported/",
                                   BehaviorTree::EFF_cpp);

const char* mainBt = "TestBehaviorGroup/dummy_test";
bool bOk = pA->btload(mainBt);
```

5.3 导出和使用 C#行为树

1. 在“导出行为树”对话框中，选择“C# Behavior Exporter”，如下图所示：



2. 点击上图中右侧的“...”设置按钮，在弹出的“C#导出设置”对话框中设置导出文件所在的位置，如下图所示：



3. 回到“导出行为树”对话框，点击“导出”按钮，开始导出.cs 文件。在指定的导出位置（默认为当前工作区的 exported 文件夹）会自动生成一个名为 generated_behaviors.cs 的文件，里面生成了在 editor 中添加的所有行为树类。

4. 将生成的 generated_behaviors.cs 文件添加进自己的游戏项目中，就可以与前面提及的 XML 行为树文件一样的接口和方式加载使用该.cs 文件。

5.4 扩展生成节点或编程语言

生成 C++或 C#的相关代码放在 Designer 项目的 PluginBehaviac 工程中，这里简单介绍用于生成 C++代码的相关类和接口（生成 C#完全类似）：

1. 从基类 Exporter 派生出子类 ExporterCpp 用于管理和发起 C++代码的生成。在模块初始化的地方调用如下代码，编辑器将会支持导出 C++代码：

   ```
   Plugin.Exporters.Add(new
   ExporterInfo(typeof(PluginBehaviac.Exporters.ExporterCpp), "cpp", "C++ Behavior
   Exporter", true, true));
   ```

2. 在 DataExporters/Cpp 文件夹中，维护了一组导出 Variable、Par、Property、Method、Enum、Struct、Array 等基本数据结构的 Utility 类。

3. 从基类 NodeExporter 派生出子类 NodeCppExporter，该子类定义了 GenerateClass()、GenerateInstance() 、 GetGeneratedClassName() 、 ShouldGenerateClass() 、GenerateConstructor()、GenerateMember()、GenerateMethod()等虚函数，用于生成某个具体的节点类。

4. 在 NodeExporters/Cpp 文件夹中，维护了各种节点类的辅助导出类，例如类 Action 对应的类 ActionCppExporter 等等，这些类分别实现了上面的虚函数，用于导出特定节点类的类型、属性和方法等信息。

6.注册自定义类、结构体和枚举

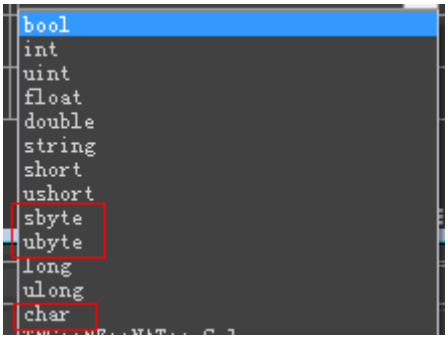
在某些情况下，除了 C++库（Lib）中提供支持的一些基本类型（bool,short,int,float 等）之外，行为树中间件还支持扩展使用已有的或者用户自定义的类、结构体和枚举类型。这里面有两种情况：

1. 有些类型是你的程序中本来就已有的，该类型或者是某个 thirdparty 的库中实现的，总之这些类型你是不方便修改的
2. 另外一些你可以随便修改的类型

根据这两种情况，处理方式是不同的。

此外，需要对 char，signed char，unsigned char 做出一些必要的说明：

1. 在 c++中，char,signed char,unsigned char 是三个不同的类型，特别的 char 和 signed char 是不同的类型
2. 在 Designer 中，分别对应 char，sbyte，ubyte



3. 对于 C#，char，sbyte，byte 是基本类型，分别对应 Designer 中的 char，sbyte 和 ubyte

6.1 扩展使用已有的类型

1.1 在.h 文件中，通过宏 **M_SPECIALIZE_TYPE_HANDLER** 特化某个需要的类型，如下代码样例所示(加入 TestNS::Float2 是某个 thirdparty 库中的类型，你需要用到但不能修改之)：
M_SPECIALIZE_TYPE_HANDLER(TestNS::Float2);

1.2 定义一个‘类似’的 struct（myFloat2），该 struct 的作用是用来定义那个已存在的类的成员，这样的话，behavaic 才能够访问该类。两个 template 的函数 GetObjectDescriptor 和 RegisterProperties 是必须的，这两个函数把这两个类‘绑定’了起来。

```
struct myFloat2
{
    float x;
    float y;
    DECLARE_BEHAVIAC_OBJECT_NOVIRTUAL(myFloat2);
    myFloat2()
    {}
    myFloat2(const TestNS::Float2& v) : x(v.x), y(v.y)
    {
    }
};

template <>
inline CTagObjectDescriptor& GetObjectDescriptor<TestNS::Float2>()
{
    return myFloat2::GetObjectDescriptor();
}

template <>
inline void RegisterProperties<TestNS::Float2>()
{
    myFloat2::RegisterProperties();
}
```

上面的 struct 的定义及两个 template 函数是定义在 header 里，下面的宏需要放在

cpp 里：

```
BEGIN_PROPERTIES_DESCRIPTION(myFloat2)
    CLASS_DISPLAYNAME(L"")
    CLASS_DESC(L"")

    REGISTER_PROPERTY(x);
    REGISTER_PROPERTY(y);
END_PROPERTIES_DESCRIPTION()
```

1.3 在命名空间StringUtils的嵌套子空间Private中实现该类型的**ToString**()和**FromString**()函数，注意myFloat2中需要实现相应的转换构造函数(myFloat2(const TestNS::Float2& v))，如下代码样例所示：

```
namespace StringUtils
{
    namespace Private
    {
        template<> inline behaviac::string_t ToString(const TestNS:: Float2& val)
        {
            myFloat2 temp(val);
            return temp.ToString();
        }
        template<> inline bool FromString(const char* str, TestNS::Float2& val)
        {
            myFloat2 temp;
            if (temp.FromString(str))
            {
                val.x = temp.x;
                val.y = temp.y;
                return   true;
            }
            return   false;
        }
    }
}
```

1.4 实现该类型的模板函数 **SwapByteTempl()**，注意该函数不能放在任何命名空间（Namespace）中，如下代码样例所示：

```
template< typename SWAPPER >
inline void SwapByteTempl(TestNS::Float2& v)
{
    SwapByteTempl< SWAPPER >(v.x);
    SwapByteTempl< SWAPPER >(v.y);
}
```

1.5 在命名空间 behaviac 的嵌套子空间 Details 中实现该类型的 Equal()模板函数，如下代码

样例所示：

```
namespace behaviac
{
    namespace Details
    {
        template<>
        inline bool Equal(const TestNS::Float2& lhs, const TestNS::Float2& rhs)
        {
            return Tag::IsEqualWithEpsilon(lhs.x, rhs.x) &&
                   Tag::IsEqualWithEpsilon(lhs.y, rhs.y);
        }
    }
}
```

1.6 在初始化注册（Register）的部分需要加上如下的代码，反注册（UnRegister）的部分添加相应 UnRegister 的代码。注意这部分 Register/UnRegister 的代码**不是必须的**，如果该类型没有用作 par 或者没有用作条件来比较的话就可以不需要。

```
Property::Register<TypeTest_t>("TypeTest_t");
Property::Register<behaviac::vector_t<TypeTest_t> >("vector<TypeTest_t>");
Condition::Register<TypeTest_t>("TypeTest_t");
Condition::Register<behaviac::vector_t<TypeTest_t> >("vector<TypeTest_t>");


Property::UnRegister<TypeTest_t>("TypeTest_t");
Property::UnRegister<behaviac::vector_t<TypeTest_t> >("vector<TypeTest_t>");
Condition::UnRegister<TypeTest_t>("TypeTest_t");
Condition::UnRegister<behaviac::vector_t<TypeTest_t> >("vector<TypeTest_t>");
```

更多细节可以参考行为树中间件源码库的 behaviac\ext\types.h 文件中对 Tag 库中的基本类型（例如类 Float2、Float3 等）的扩展使用，及 behaviorRegister.cpp 中的 Register 和 UnRegister。

当然了，也可以参考下面的自定义类或结构体来定义一个结构体来包装和转换那个已存在的类型。例如如果用到了类型 D3DVector，除了用这里介绍的几个步骤来通知 behaviac 使用外，也可以通过下面的自定义类或结构体定义 myD3DVector 来包装和转换 D3DVector，你的所有 behaviac 相关代码则使用 myD3DVector。

std::string 已经支持，可以直接使用。不过，最好使用 behaviac::string_t，因为 behaviac::string_t 使用了定制的 allocator，可以对内存的使用进行统一的管理。

6.2 自定义类或结构体

1.7 在 .h 文件中，任意编写一个自定义的类或结构体，并用宏 **DECLARE_BEHAVIAC_OBJECT_NOVIRTUAL** 声明该类或者结构体为非虚类，如下代码样例所示：

```
struct TypeTest_t
{
    int         name;
    float       weight;
    bool        bLive;

    DECLARE_BEHAVIAC_OBJECT_NOVIRTUAL(TypeTest_t);
};
```

1.8 在.cpp 文件中，通过一系列宏注册该类或结构体自身的描述及其属性：

1.8.1 宏 **BEGIN_PROPERTIES_DESCRIPTION** 和 **END_PROPERTIES_DESCRIPTION** 表示类型信息注册的开始和结束。

1.8.2 可选的宏 **CLASS_DISPLAYNAME** 和 **CLASS_DESC** 用于注册类或结构体自身的显示名和描述。

1.8.3 宏 **REGISTER_PROPERTY** 用于注册类或结构体的属性，可以通过**.DISPLAYNAME** 的追加方式为属性添加显示名，通过**.DESC** 的追加方式为属性添加描述。

如下代码样例所示：

```
BEGIN_PROPERTIES_DESCRIPTION(TypeTest_t)
    CLASS_DISPLAYNAME(L"测试结构体")
    CLASS_DESC(L"自定义结构体")

    REGISTER_PROPERTY(name);
    REGISTER_PROPERTY(weight).DISPLAYNAME(L"重量");
    REGISTER_PROPERTY(bLive) .DISPLAYNAME(L"是否活着").DESC(L"存活状态");
END_PROPERTIES_DESCRIPTION()
```

1.9 在初始化注册（Register）的部分需要加上如下的代码，反注册（UnRegister）的部分添加相应 UnRegister 的代码。注意这部分 Register/UnRegister 的代码**不是必须的**，如果该类型没有用作 par 或者没有用作条件来比较的话就可以不需要（参考 4 struct、enum 的数组）。

```
Property::Register<TypeTest_t>("TypeTest_t");
Property::Register<behaviac::vector_t<TypeTest_t> >("vector<TypeTest_t>");
Condition::Register<TypeTest_t>("TypeTest_t");
Condition::Register<behaviac::vector_t<TypeTest_t> >("vector<TypeTest_t>");


Property::UnRegister<TypeTest_t>("TypeTest_t");
Property::UnRegister<behaviac::vector_t<TypeTest_t> >("vector<TypeTest_t>");
Condition::UnRegister<TypeTest_t>("TypeTest_t");
Condition::UnRegister<behaviac::vector_t<TypeTest_t> >("vector<TypeTest_t>");
```

更多细节可以参考行为树中间件源码库中 btunittest 工程的 behaviortest.h 和 behaviorRegister.cpp 文件中 TypeTest_t 结构体相关的代码。

6.3 自定义枚举类型

6.1 类似上面的自定义类，在.h 文件中，任意编写一个自定义枚举类型，然后通过宏 **DECLARE_BEHAVIAC_OBJECT_ENUM** 声明该枚举类型。需要注意的是，宏 **DECLARE_BEHAVIAC_OBJECT_ENUM** 必须定义在全局的 namespace 里，即放在任何命名空间之外。

如下代码样例所示：

```
namespace TNS
{
  enum EnumTest
  {
      ET_red,
      ET_green,
      ET_blue
  };
}

DECLARE_BEHAVIAC_OBJECT_ENUM(TNS::EnumTest, EnumTest);
```

6.2 在.cpp 文件中，通过一系列宏注册该枚举自身的描述及其枚举值：

6.2.1 宏 **BEGIN_ENUM_DESCRIPTION** 和 **END_ENUM_DESCRIPTION** 表示枚举信息注册的开始和结束。

6.2.2 可选的宏 **ENUMCLASS_DISPLAYNAME** 和 **ENUMCLASS_DESC** 用于注册枚举自身的显示名和描述。

6.2.3 宏 **DEFINE_ENUM_VALUE** 用于注册枚举的值，可以通过**.DISPLAYNAME** 的追加方式为值添加显示名，通过**.DESC** 的追加方式为值添加描述。

如下代码样例所示：

```
BEGIN_ENUM_DESCRIPTION(TNS::EnumTest, EnumTest)
    ENUMCLASS_DISPLAYNAME(L"枚举EnumTest");
    ENUMCLASS_DESC(L"枚举EnumTest的说明");

    DEFINE_ENUM_VALUE(ET_red, "Red");
    DEFINE_ENUM_VALUE(ET_green, "Green").DISPLAYNAME(L"绿色");
    DEFINE_ENUM_VALUE(ET_blue, "Blue").DISPLAYNAME(L"兰色").DESC(L"兰值");
END_ENUM_DESCRIPTION()
```

6.3 在初始化注册（Register）的部分需要加上如下的代码，反注册（UnRegister）的部分添加相应 UnRegister 的代码。注意这部分 Register/UnRegister 的代码**不是必须的**，如果该类型没有用作 par 或者没有用作条件来比较的话就可以不需要（参考 4 struct、enum 的数组）。

```
Property::Register<TypeTest_t>("TypeTest_t");
Property::Register<behaviac::vector_t<TypeTest_t> >("vector<TypeTest_t>");
Condition::Register<TypeTest_t>("TypeTest_t");
Condition::Register<behaviac::vector_t<TypeTest_t> >("vector<TypeTest_t>");

Property::UnRegister<TypeTest_t>("TypeTest_t");
Property::UnRegister<behaviac::vector_t<TypeTest_t> >("vector<TypeTest_t>");
Condition::UnRegister<TypeTest_t>("TypeTest_t");
Condition::UnRegister<behaviac::vector_t<TypeTest_t> >("vector<TypeTest_t>");
```

更多细节可以参考行为树中间件源码库中 btunittest 工程的 behaviortest.h 和 behaviorRegister.cpp 文件中 EnumTest 枚举相关的代码。

6.4 struct、enum 的数组

基本类型（bool，int，float，char，sbyte，ubyte 等）的数组可以直接使用。

但当使用到其他自定义类型数组的时候，需要添加特殊的宏和代码，否则，运行时可能会有错误（如果没有使用到相应的操作的话没有错误）：

1. 在.h 的头文件里添加如下所示的宏

```
BEHAVIAC_OVERRIDE_TYPE_NAME(behaviac::vector<test_ns::TypeTest_t>);
SPECIALIZE_TYPE_HANDLER(behaviac::vector<test_ns::TypeTest_t>, BasicTypeHandler< behaviac::vector<test_ns::TypeTest_t> >);
```

2. 在初始化的代码里添加如下所示的注册代码

```
Property::Register<test_ns::TypeTest_t>("test_ns::TypeTest_t");
Property::Register<behaviac::vector<test_ns::TypeTest_t> >("vector<test_ns::TypeTest_t>");
Condition::Register<test_ns::TypeTest_t>("test_ns::TypeTest_t");
Condition::Register<behaviac::vector<test_ns::TypeTest_t> >("vector<test_ns::TypeTest_t>");
```

3. 在结束代码里添加如下所示的反注册代码

```
Property::UnRegister<test_ns::TypeTest_t>("test_ns::TypeTest_t");
Property::UnRegister<behaviac::vector<test_ns::TypeTest_t> >("vector<test_ns::TypeTest_t>");
Condition::UnRegister<test_ns::TypeTest_t>("test_ns::TypeTest_t");
Condition::UnRegister<behaviac::vector<test_ns::TypeTest_t> >("vector<test_ns::TypeTest_t>");
```

6.5 Agent 及其子类的数组

当涉及 behaviac::Agent 或是其子类的时候，仅支持其指针类型即 behaviac::Agent*或 SubclassAgent*（SubclassAgent 是 behaviac::Agent 的一个子类）。

behaviac::Agent*或 vector<behaviac::Agent*>类型直接被支持，不需要做什么额外的事情。其任何子类 SubclassAgent*也直接被支持。

但是需要支持 vector<SubclassAgent*>的时候，则需要在.h 文件里添加：

```
BEHAVIAC_OVERRIDE_TYPE_NAME(behaviac::vector< SubclassAgent *>);
SPECIALIZE_TYPE_HANDLER(behaviac::vector< SubclassAgent *>, BasicTypeHandler< behaviac::vector<
```

```
SubclassAgent *> >);
```

在初始化和结束的时候分别注册和反注册（请注意前面的 SubclassAgent 是有*的而后面“”里的 SubclassAgent 没有*）：

```
behaviac::Property::Register<behaviac::vector<SubclassAgent*> >("vector<SubclassAgent>");
behaviac::Condition::Register<behaviac::vector< SubclassAgent *> >("vector< SubclassAgent>");

behaviac::Property::UnRegister<behaviac::vector<SubclassAgent*> >("vector<SubclassAgent>");
behaviac::Condition::UnRegister<behaviac::vector< SubclassAgent *> >("vector< SubclassAgent>");
```

7 扩展节点类型

有时候，behaviac 自带的节点类型不能满足项目需求，或者需要更高效的节点类型。可以通过添加新的节点类型来扩展系统。添加新的节点类型需要实现一个 **C#的 designer 端的 plugin**，以及 **C++端需要实现相应的节点类型**来 load 数据并且负责 update。

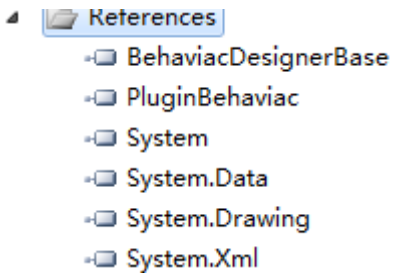
https://github.com/TencentOpen/behaviac

7.1 编辑器端的 plugin

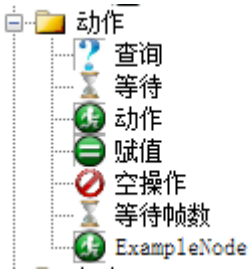
随项目提供了 ExamplePlugin，用户可以根据该 plugin 做相应的修改来添加新的节点。该项目 Path：behaviac\tools\Designer\Plugins\ExamplePlugin

7.2 项目设置

1. 该项目依赖如下 dlls，如下所示：



2. 产生的 dll 需要放置在 BehaviacDesigner.exe 同一个目录。
3. 如果你直接备份 ExamplePlugin 只是改了相应的名字，并且该 plugin 的项目和 ExamplePlugin 的位置（Path）一样，即位于 behaviac\tools\Designer\Plugins\，那么对 BehaviacDesignerBase.dll 的引用以及产生 dll 的 OutPath 不需要改动即可。否则如果有编译错误或者运行时不能看到你添加的节点，请参考 1 和 2 确保都是正确的。
4. 如图一起正常的话，打开 designer 后，你新加的节点 ExampleNode 将出现在 Action 组下面。



7.3 节点类型

每一个要添加的节点类型，都需要仿照 ExampleNode 或相应的节点新建文件 ExampleNode.cs，并编写类 ExampleNode 的实现，如下代码所示：

```
[NodeDesc("Actions", NodeIcon.Action)]
public class ExampleNode : Behaviac.Design.Nodes.Action
{
    public ExampleNode()
        : base(Resources.ExampleNode, Resources.ExampleNodeDesc)
    {
    }

    public override string ExportClass
    {
        get { return "ExampleNode"; }
    }
```

1. 该类的 Attribute（即 NodeDesc）用来指定该节点出现的组和图标（icon）。如果使用自己的图标文件，则需要添加进工程的 Resources.resx 文件中，假如该图标命名为“exampleNodeIcon”，并使用如下代码：
   [NodeDesc("Actions", “exampleNodeIcon”)]
2. 构造函数里需要提供显示用的名字（DisplanName）和描述（Description）
3. ExportClass 用来指定该类导出供 C++载入时的类名。
4. 根据需要添加自己的属性（Property），这些属性在 designer 里将会自动的显示出来供用户指定数据。

```
private VariableDef _property;
[DesignerPropertyEnum("OperandRight", "OperandRightDesc", "Condition", DesignerProperty.I
public VariableDef Property
```

5. 如果上一步添加了属性，那么需要重载 protected override void CloneProperties(Node newnode)，来 clone 这些新加的属性。
6. 重载 public override void CheckForErrors(BehaviorNode rootBehavior, List<ErrorCheck> result)来验证数据的有效性已经提供无效时的说明。
7. 此外，还需编写生成新加节点类型的 C++或 C#相关的代码，参考“5.4 扩展生成节点或编程语言”。这里介绍用于生成 C++代码的相关类和接口（生成 C#完全类似）：
   1) 添加文件 ExampleNodeCppExporter.cs（注意，必须是 CppExporter 结尾，若是生成 C#，则必须以 CsExporter 结尾，后面类推），编写类 ExampleNodeCppExporter 的实现。
   2) 根据新加节点的需要，重载 GenerateConstructor、GenerateMember、GenerateMethod 等函数。
   3) 其他细节请参考样例代码的实现。

7.4 属性

1. 属性 Property 的类型可以是基本类型如 bool、int、float、string 等
2. 可以是 VariableDef、MethodDef，RightValueDef 来表示访问 Agent 的属性或方法。
3. 可以是 struct，数组或 struct 的数组。
4. 可以是枚举 enum。
5. 需要提供相应的 Attribute 来描述该属性的 meta 信息，诸如，显示名字、描述、范围、显示顺序等。请注意属性 Property 的类型必须与相应的 attribute 的类型一致。

- DesignerBoolean
- DesignerInteger
- DesignerFloat
- DesignerString
- DesignerTypeEnum
- DesignerStruct
- DesignerPropertyEnum
- DesignerRightValueEnum
- DesignerMethodEnum
- DesignerArrayStruct
- DesignerArrayBoolean
- DesignerArrayInteger
- DesignerArrayFloat
- DesignerArrayString
- DesignerArrayEnum

7.5 运行时库端需要实现相应的节点类型

节点类型可以是子节点（Leaf），或者是分支节点，而分支节点可以是组合节点（Composite），或者是装饰节点（Decorator）。需要根据具体的节点类型从相应的基类继承来创建自己的节点类。每个节点类型需要两个相应的 class，一个是**静态 Node**，这个是静态的数据，负责 load 数据以及创建相应的动态节点，另一个是**动态 Task**。请参考 Action.h 或 Condition.h 或 waitframes.h 等头文件。

7.5.1 静态 Node

静态的 Node 必须提供的两个虚函数如下：

1. `virtual void load(int version, const char* agentType, const properties_t& properties);`
2. 和 `virtual BehaviorTask* createTask() const;`
3. `BehaviorNode::Register<WaitFrames>();`进入程序前需要注册该静态的 Node 类型，动态的 Task 类不需要注册。
4. `BehaviorNode::UnRegister<WaitFrames>();`结束程序前需要反注册该静态的 Node 类型，动态的 Task 类不需要反注册。

```
class BEHAVIAC_API WaitFrames : public BehaviorNode
{
public:
    BEHAVIAC_DECLARE_DYNAMIC_TYPE(WaitFrames, BehaviorNode);

    WaitFrames();
    virtual ~WaitFrames();
    virtual void load(int version, const char* agentType, const properties_t& properties);

    int GetFrames(Agent* pAgent) const;

private:
    virtual BehaviorTask* createTask() const;

private:
    Property*       m_frames_var;
    CMethodBase*    m_frames_method;

    friend class WaitFramesTask;
};
```

7.5.2 动态 Task

对应静态 Node，动态 Task 需要提供下列虚函数：

1. `virtual void copyto(BehaviorTask* target) const;`
2. `virtual void save(ISerializableNode* node) const;`
3. `virtual void load(ISerializableNode* node);`
4. `virtual bool onenter(Agent* pAgent);`
5. `virtual void onexit(Agent* pAgent, EBTStatus s);`
6. `virtual EBTStatus update(Agent* pAgent, Interval_t& interval, EBTStatus childStatus);`
7. 没有特殊需要的话调用 super 的函数就可以了。

```
class BEHAVIAC_API WaitFramesTask : public LeafTask
{
public:
    BEHAVIAC_DECLARE_DYNAMIC_TYPE(WaitFramesTask, LeafTask);

    WaitFramesTask();
protected:
    virtual ~WaitFramesTask();

    virtual void copyto(BehaviorTask* target) const;
    virtual void save(ISerializableNode* node) const;
    virtual void load(ISerializableNode* node);

    virtual bool onenter(Agent* pAgent);
    virtual void onexit(Agent* pAgent, EBTStatus s);
    virtual EBTStatus update(Agent* pAgent, Interval_t& interval, EBTStatus childStatus);

    int GetFrames(Agent* pAgent) const;
private:
    int     m_start;
    int     m_frames;
};
```

8 优化及性能等

宏 `BEHAVIAC_RELEASE` 定义的时候是最终版，`BEHAVIAC_RELEASE` 没有定义的时候是为开发版。

在 C++下，`BEHAVIAC_RELEASE` 在 Debug 和 Profiling 版本中没有定义，在 Release 版本中有定义。最终版本中，behaviac::Agent::ExportMetas 和 behaviac::Socket::SetupConnection 等函数将什么都不做。当然在某些情况下，比如需要在 Release 版本中也做出某些测试的时候，也可以选择在项目文件中定义 `BEHAVIAC_RELEASE`。

在开发版中，为了支持调试，特别是连接编辑器的调试，logging 和 socketing 是打开的，由于有文件操作，运行效率将受到极大的影响。可以通过 behaviac::Config::IsLogging 和 behaviac::Config::IsSocketing 来控制是否要 Log 到文件或是否和编辑器的连接。

而当 `BEHAVIAC_RELEASE` 定义的时候的最终版里，logging 和 socketing 是关闭的，也不支持连接编辑器。

总之，针对效率可以有下述选择：

C++下，

1. 选择 Release 版，没有调试功能但效率最高
2. 如果需要调试功能，则需要选择 Debug 或 Profiling 版，这个时候，可以通过 behaviac::Config::SetLogging 和 behaviac::Config::SetSocketing 来控制是否打开 logging 和 socketing。

C#下，

1. 在 Assets 目录下的 smcs.rsp 文件中，定义 `BEHAVIAC_RELEASE`，没有调试功能但效率最高（当然也可以选择其他方式定义 `BEHAVIAC_RELEASE` 来打开最终版）。
2. 如果需要调试功能，则不能定义 `BEHAVIAC_RELEASE`，这个时候，可以通过 behaviac.Config.IsLogging 和 behaviac.Config.IsSocketing 来控制是否打开 logging 和 socketing。

9 Revision

2014 年 8 月 20 日

4 struct、enum 的数组

5 Agent 及其子类的数组

2014 年 4 月 29 日

行为树编辑器的使用

2 注册、导出和使用元信息

3 导出行为树

4 注册自定义类、结构体和枚举

5 扩展节点类型

2013 年 3 月 29 日

初始版本